9/Aug/23

MIT 6.01SC

Introduction to Electrical Engineering and Computer Science

# Unit 1

=> Humans deal with complexity by exploiting the power of abstraction & modularity.

Modularity → Build components that can be reused.

Abstraction → 1) Once module is built, only specific ~~simpler~~ description needed to work with it ahead.

2) Most of the details can be ignored.

=> Complicated Design Problem

↓

P C A P

Primitive → basic sets

combination → combine

abstraction → packing to form new primitives

Pattern → Pattern of abstracting our abstraction

=> Good ~~modules~~ modules preserve the barrier of abstraction b/w the use of module.

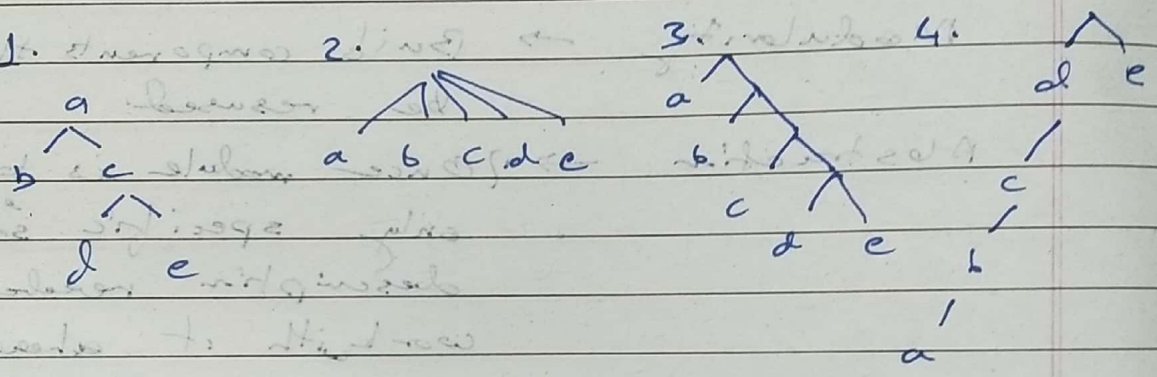
P C A P

↓ ↓ ↓ → what patterns to look for.

what are the primitives | How do you combine them | How do you abstract the behaviors

eg: list [a, [b, [c], [d, e]]]

1. 2. 3. 4.




Ans: 3

Vertices are list!
Hirierchical part.

→ Composition allows hieraehical construction of complex operation.

→ $(x^3+x^2+x+1)(4x+1) =$

$(4x^3 + 4x^2 + 4x + 4x + [0]$ * order

$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

14/Aug/23

# State Machines < Continous / discontinous

↓

method of modeling Systems

↓

output depends on { history of previous inputs.

Different ways to structure a Progam

3 ways

1] Imperative (Procedural)

↳ step by step instructions

eg. cooking with recepie

take this, add this, etc

2] Functional Program

↳ focus on procedures. such as mathematical funtins i/p ⟹ o/p

↳ not chge outside variables. That we may need further

3) Object oriented Program

→ focus of building collection of data with procedure that are related

→ organizing solution in tems of hirerarchy of these structures.

Q What is minimum length of seq of increment & square operations needed to transform 1 to 100?

1: <4 2: 4 3: 5 4: 6 5: >6

Ans) 3: 5

1 → 2 inc

2 → 3 inc

3 → 9 sq

9 → 10 inc

10 → 100 sq

1) Procedural ways

→ think about sequences and order them by length.

eg) find Sequence (1, 100)
enumerate over length of seq. that are possible.

length of seq.

1: (1 inc, 2)
1: (1 sq., 1)
2: (1 inc inc, 3)
2: (1 inc sq, 4)
⋮

```
def findSeq(initial, goal):
    # track of all diff seq. found.
    # as list of tuples.
    candidates = [(str(initial), initial)]
    for i in range(1, goal - initial + 1):
        new_Candidates = []
        for action, result in candidates:
            for a, r in [('inc', inc), ('sq', sq)]:
                new_Candidates.append(action+a, r(result))
        if new_cand[-1][1] == goal
            return
        candidates = new_Candidates
```

2) Functional Program
↳ focus on procedure.
↓
that implement function of mathematic type.

apply - pure function
↓
cact some i/ps
↳ generate o/p
↳ has no side effects

```
def apply (opList, arg):
    if len(opList) == 0
        return arg
    else:
        return (opList[1:], opList[0](arg))
```

Procedure - First Class Object.
↳ making list of fuctions (procedure)

addLevel (opList, funcList)
↳ everytime create a new list of list with double the no of elements as old list of list
↳ has 2 possibilities [ a) addby sq. part
b) addby inc. operator

```
def addLevel(optList, fctList):
    return [x+[y] for y in fctList for x in optList]
```

⇒ recursion is good ways to reduce complicated → simpler

eg

$$f(n) = \begin{cases} 1 & \text{if } n=0 \\ b\,f(n-1) & \text{if } n>0 \end{cases}$$

but if doing raising num to 1024 power.

speed up

$$f(n) = \begin{cases} 1 & \text{if } n=0 \\ b\,f(n-1) & \text{if } n \text{ odd} \\ (f(n/2))^2 & \text{otherwise} \end{cases}$$

Q How many invocation of fastExpoent is generated by fastExpoent(2,10):

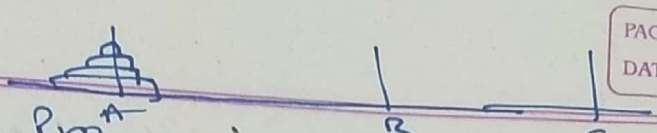
this is call to function

2,10 2,5 2,4 2,2 2,1 2,0

there 5 are invocation

my ans: 6

Correct Ans: 5




⇒ Function Programming

↳ is Expressive

↓

way to incorporate knowledge about an answer into a solution

Complicated ⟶ simple

eg Tower of Hanoi

→ Power of Recursion

↳ is in thinking

→ Hard problem

↳ reduced to problem of same type just simpler.

```
def Hanoi (n, A, B, C):
    if n == 1:
        print 'move from' + A + to + B'

    else:
        Hanoi (n-1, A, C, B)
        Hano: ( 1, A, B, C)
        Hanoi (n-1, C, B, A)
```

→ take top (n-1) from A to C

↳ How? don't know yet!

→ take bottom to B

→ take the (n-1) from C to B

↳ How? don't know!

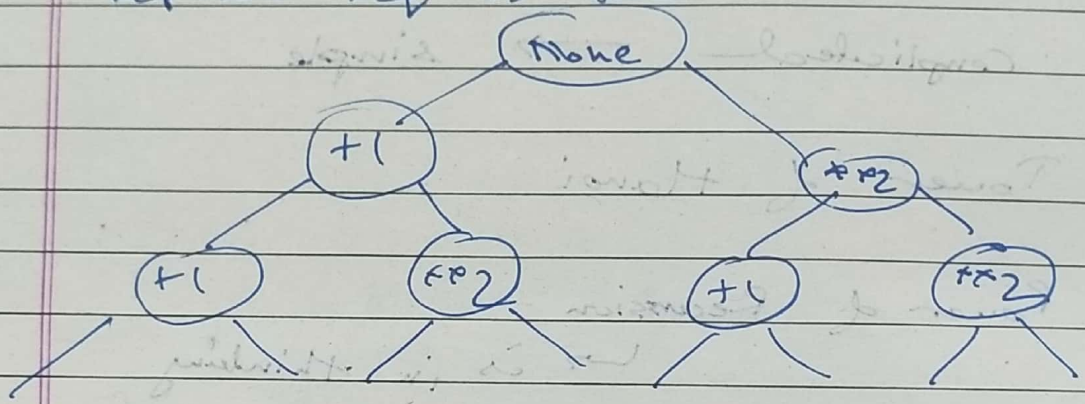
↳ Just recurse till get n=1.

9/Aug/23

Recursive solution are expressive.

3) Object Oriented Approach
↳ changed the representation

Represent seq as tree




Idea → think of a useful object
↳ what would I like to remember?

```
class Node:
    def __init__(self, parent, action, answer):
        self.parent = parent
        self.action = action
        self.answer = answer
    def path(self):
        if self.parent == None:
            return [(self.action, self.answer)]
        else:
            return self.parent.path()
                + [(self.action, self.answer)]
```

19/Aug/23

```
def findSequence(initial, goal):
    q = [Node(None, None, 1)]
    while q:
        parent = q.pop()
        for (a, r) in [("inc", inc), ("sq", sq)]:
            newNode = Node(parent, a, r(parent.answer))
            if newNode.answer == goal
                return newNode.path()
            else
                q.append(newNode)
    return None.
```

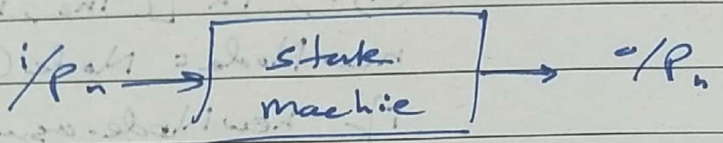
⇒ Todo → build a represent of solution out of pieces that make sense for this problem
→ think about modularity at higher level.

The way you structure program —influence→ significantly how modular solution

# Process ⇒ is some set of procedure that evolve over time.
there is not an answer
the answer evolves overtime

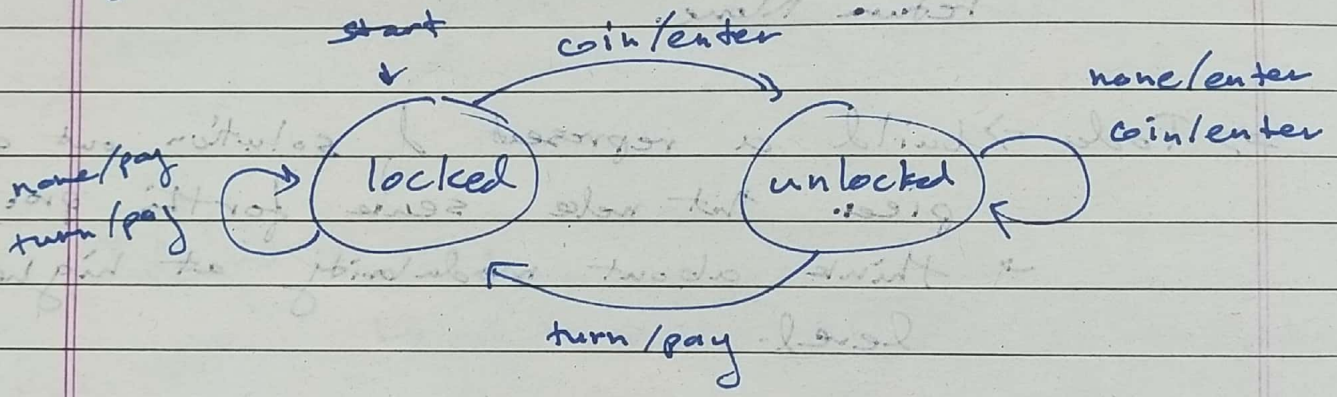
# State Machine

↳ very convinient structure to capture behaviour of process




Nodes represent process

eg) Turnstil




State transition Diagram

Behaviours → set of o/p as function of time.

Idea of State Machine is modular

Stat machine in Python

1) SM Class
   ↳ start  # how to start
   step
   transduce  # repeated call to steps
   ⇓
   Combine seq. of i/p into
   seq. of o/p's

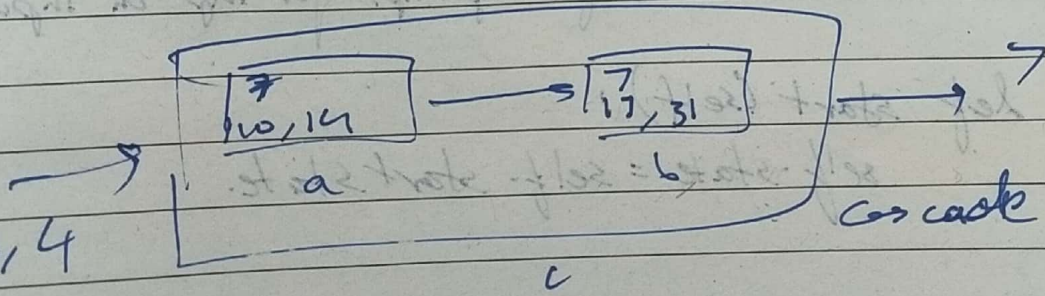
2) Turnstile Class
   ↳ sub class
   start state
   getNextState

3) Turnstile Instace

start & step are necessary routing for SMs

Q

c.transduce([7, 3, 4])




Ans: 3) 7, 17, 3

20/Aug/23

# State Machine in Python

class SM:
↳ attribute : startState
↳ method : getNextValues (is a pure function — does not mutate state variable.)

$(s_t, i_t) \rightarrow (s_{t+1}, o_t)$

eg class Accumulator(SM):
    startState = 0
    def getNextValues(self, state, input):
        return (state + input, state + input)

Running the SM.

```
def step(self, inp):
    (s, o) = self.getNextValue(self.state, inp)
    self.state = s
    return o
def transduce(self, inputs):
    self.start()
    return [self.step(inp) for inp in inputs]
def start(self):
    self.state = self.startState
```

optional method: getNextState

↓ returns

[ value which is both state & o/p ]

∴ used in getNextValues

```
def getNextValues(self, state, inp):
    nextState = self.getNextState(state, inp)
    return (nextState, nextState)

def getNextState(state, inp):
    return state + inp.
```

o/p of getNextState → next state, output